2024/10/22 区间、排序、单调栈

Updated 1050 GMT+8 Oct 22, 2024

2024 fall, Complied by Hongfei Yan

> Log:
>
> 练习网址，https://leetcode.cn 。leetcode只需要完成核心代码就可以，不用写输入输出部分。
>
> 核心代码已经指定好类名、方法名、参数名，请勿修改或重命名，直接返回值即可。

一、Recap

> 取自 20241015_greedy_matrices.md

贪心算法是用来解决一类最优化问题，并希望由局部最优策略来推得全局最优结果。贪心法适用的问题一定满足最优子结构性质，即一个问题的最优解可以通过其子问题的最优解来构建。

严谨使用贪心法来求解最优问题需要对采取的策略进行证明。证明往往比贪心本身更难，因此一般来说，如果想到某个似乎可行的策略，并且自己无法举出反例，那么就编码实现尝试。

> https://oi-wiki.org/basic/greedy/
>
> 贪心算法有两种证明方法：反证法和归纳法。一般情况下，一道题只会用到其中的一种方法来证明。
>
> 1. 反证法：如果交换方案中任意两个元素/相邻的两个元素后，答案不会变得更好，那么可以推定目前的解已经是最优解了。
> 2. 归纳法：先算得出边界情况（例如 `n=1`）的最优解 F_1），然后再证明：对于每个 n，F_{n+1} 都可以 由 F_n 推导出结果。
>
> **常见题型**
>
> 在提高组难度以下的题目中，最常见的贪心有两种。
>
> - 「我们将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择。」。
> - 「我们每次都取 XXX 中最大/小的东西，并更新 XXX。」（有时「XXX 中最大/小的东西」可以优化，比如用优先队列维护）
>
> 二者的区别在于一种是离线的，先处理后选择；一种是在线的，边处理边选择。
>
> **排序解法**
>
> 用排序法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模拟计算的方法求出最优值。
>
> **后悔解法**
>
> 思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。
>
> **与动态规划的区别**
>
> 贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

双指针和二分查找

> 取自 20241015_greedy_matrices.md。

双指针和二分查找是贪心算法中常用的技巧。常规贪心题目，例如：

1 two pointers

参考《算法笔记.胡凡》4.6

two pointers 是算法编程中一种非常重要的思想，但是很少会有教材单独拿出来讲，其中一个原因是它更倾向于是一种编程技巧，而长得不太像是一个“算法”的模样。two pointers的思想十分简洁，但却提供了非常高的算法效率。

以一个例子引入：给定一个递增的正整数序列和一个正整数 M，求序列中的两个不同位置的数a和 b，使得它们的和恰好为 M，输出所有满足条件的方案。例如给定序列{1,2,3,4,5.6}和正整数M=8，就存在2+6=8与3+5=8成立。本题的一个最直观的想法是，使用二重循环枚举序列中的整数a和 b，判断它们的和是否为 M，如果是，输出方案；如果不是，则继续枚举。代码如下:

```
n = int(input())
a = list(map(int, input().split()))
M = int(input())

for i in range(n):
    for j in range(i + 1, n):
        if a[i] + a[j] == M:
            print(a[i],a[j])
"""
5
1 2 3 4 5
7

2 5
3 4
"""
```

显然，这种做法的时间复杂度为 O(n^2)，对n在 10^5的规模时是不可承受的。

two pointers 将利用有序序列的枚举特性来有效降低复杂度。它针对本题的算法过程是：令下标i的初值为 0，下标j的初值为n-1，即令i、j分别指向序列的第一个元素和最后一个元素，接下来根据 a[i] + a[j]与M 的大小来进行下面三种选择，使i不断向右移动、使j不断向左移动，直到i>j成立。

① 如果满足 a[i] + a[] ==M，说明找到了其中一组方案。由于序列递增，不等式 a[i+1]+a[j]>M 与 a[i] + a[j-1]<M均成立，但是 a[i+1]+a[j-1]与M 的大小未知，因此剩余的方案只可能在[i+1,j-1]区间内产生，令i=i+1、j=j-1（即令i向右移动，j 向左移动）。

② 如果满足 a[i] + a[j]>M,由于序列递增，不等式 a[i+1]+ a[j]>M 成立，但是 a[i]+ a[j-1]与M 的大小未知，因此剩余的方案只可能在[i,j-1]区间内产生，令j=j-1（即令j向左移动）。
③ 如果满足 a[i]+a[j]<M，由于序列递增，不等式 a[i]+ a[j-1]<M 成立，但是 a[i+1]+a[j]与M 的大小未知，因此剩余的方案只可能在[i+1,j]区间内产生，令i=i+1（即令i向右移动）。
反复执行上面三个判断，直到i>i成立。代码如下:

```
n = int(input())
a = list(map(int, input().split()))
M = int(input())

i = 0
j = n - 1

```

```
while i < j:
    if a[i] + a[j] == M:
        print(a[i], a[j])
        i += 1
        j -= 1
    elif a[i] + a[j] < M:
        i += 1
    else:
        j -= 1

"""
5
1 2 3 4 5
7

2 5
3 4
"""
```

分析算法的复杂度，由于i的初值为 0，j的初值为 n-1，而程序中变量i只有递增操作、变量j只有递减操作，且循环当i>i时停止，因此i和i的操作次数最多为n次，时间复杂度为 O(n)。可以发现，two pointers 的思想充分利用了序列递增的性质，以很浅显的思想降低了复杂度。

再来看序列合并问题。假设有两个递增序列A 与B，要求将它们合并为一个递增序列C。同样的，可以设置两个下标i和j，初值均为 0，表示分别指向序列 A 的第一个元素和序列B的第一个元素，然后根据 A[i]与 B[i]的大小来决定哪一个放入序列 C。

① 若 A[i] < B[i]，说明 A[i]是当前序列 A 与序列 B 的剩余元素中最小的那个，因此把A[i]加入序列 C 中，并让i加1（即让i右移一位）。

② 若 A[i] > B[j]，说明 B[i]是当前序列 A 与序列B 的剩余元素中最小的那个，因此把B[i]加入序列C 中，并让j加1（即让j右移一位）。

③ 若 A[i] == B[j]，则任意选一个加入到序列 C 中，并让对应的下标加 1。上面的分支操作直到i、j中的一个到达序列末端为止，然后将另一个序列的所有元素依次加入序列C 中，代码如下:

```
def merge(A, B):
    i, j = 0, 0
    c = []

    # 合并两个有序数组
    while i < len(A) and j < len(B):
        if A[i] <= B[j]:
            c.append(A[i])
            i += 1
        else:
            c.append(B[j])
            j += 1

    # 将 A 的剩余元素加入 c
```

```
    c.extend(A[i:])

    # 将 B 的剩余元素加入 c
    c.extend(B[j:])

    return len(c), c

# 示例
A = [1, 3, 5, 7]
B = [2, 4, 6, 8]

length, c = merge(A, B)
print(c)
```

two pointers 到底是怎样的一种思想？事实上，two pointers 最原始的含义就是针对本节第一个问题而言的，而广义上的 two pointers 则是利用问题本身与序列的特性，使用两个下标i、j对序列进行扫描（可以同向扫描，也可以反向扫描），以较低的复杂度（一般是 O(n)的复杂度）解决问题。在实际编程时要能够有使用这种思想的意识。

2 Binary Search

查找操作是编程中的基本技能，根据数据集的大小和结构选择合适的查找方法可以显著提高效率。线性查找适用于较小或无序的数据集，而二分查找适用于较大的有序数据集。

我发现二分查找容易理解，但是细节部分不容易写对（while的条件是<=，还是<；折半后是mid+1，mid-1，还是mid）。

> **常见的查找方法**
>
> 1. 线性查找（Linear Search）：
>   - 适用范围：适用于较小的数据集或无序的数据集。
>   - 原理：逐个检查数据集中的每个元素，直到找到满足条件的元素或遍历完所有元素。
>   - 时间复杂度：O(n)，其中 n 是数据集的大小。
> 2. 二分查找（Binary Search）：
>   - 适用范围：适用于有序的数据集。
>   - 原理：通过将数据集分成两半，逐步缩小查找范围，直到找到满足条件的元素或确定不存在。
>   - 时间复杂度：O(log n)，其中 n 是数据集的大小。
>
> **示例代码**
>
> 线性查找

```python
def linear_search(arr, target):
    for i, element in enumerate(arr):
        if element == target:
            return i  # 返回目标元素的索引
    return -1  # 如果未找到目标元素，返回 -1

# 示例
arr = [3, 5, 2, 8, 1, 9, 4]
target = 8
result = linear_search(arr, target)
print(f"Target {target} found at index {result}")
# Target 8 found at index 3
```

二分查找

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid  # 返回目标元素的索引
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1  # 如果未找到目标元素，返回 -1

# 示例
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
target = 8
result = binary_search(arr, target)
print(f"Target {target} found at index {result}")
# Target 8 found at index 7
```

详细步骤

线性查找

1. 初始化：
    - 遍历数据集中的每个元素。
2. 查找过程：
    - 逐个检查每个元素是否等于目标元素。
    - 如果找到目标元素，返回其索引。
    - 如果遍历完所有元素仍未找到目标元素，返回 -1。

二分查找

1. 初始化：
    - 设置左边界 `left` 为 0，右边界 `right` 为数据集的最后一个索引。
2. 查找过程：
    - 计算中间位置 `mid`。
    - 如果中间位置的元素等于目标元素，返回其索引。
    - 如果中间位置的元素小于目标元素，调整左边界 `left` 为 `mid + 1`。
    - 如果中间位置的元素大于目标元素，调整右边界 `right` 为 `mid - 1`。
    - 重复上述步骤，直到找到目标元素或左边界超过右边界。
3. 未找到目标元素：
    - 如果左边界超过右边界，返回 -1。

参考 bisect 源码的二分查找写法，

https://github.com/python/cpython/blob/main/Lib/bisect.py

当然可以！下面是一个基于 `bisect_left` 函数的实现，并提供一个详细的二分查找样例。

二分查找实现

```
def bisect_left(a, x, lo=0, hi=None, *, key=None):
"""Return the index where to insert item x in list a, assuming a is sorted.

The return value i is such that all e in a[:i] have e < x, and all e in
a[i:] have e >= x.  So if x already appears in the list, a.insert(i, x) will
insert just before the leftmost x already there.

Optional args lo (default 0) and hi (default len(a)) bound the
slice of a to be searched.

A custom key function can be supplied to customize the sort order.
"""

if lo < 0:
raise ValueError('lo must be non-negative')
if hi is None:
hi = len(a)
# Note, the comparison uses "<" to match the
# __lt__() logic in list.sort() and in heapq.
if key is None:
while lo < hi:
    mid = (lo + hi) // 2
    if a[mid] < x:
        lo = mid + 1
    else:
        hi = mid
```

```
else:
while lo < hi:
    mid = (lo + hi) // 2
    if key(a[mid]) < x:
        lo = mid + 1
    else:
        hi = mid
return lo

# 二分查找函数
def binary_search(arr, target):
index = bisect_left(arr, target)
if index != len(arr) and arr[index] == target:
return index  # 返回目标值的索引
else:
return -1  # 如果未找到目标值，返回 -1

# 示例
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
target = 8
result = binary_search(arr, target)
print(f"Target {target} found at index {result}")
```

详细步骤

1. **定义 `bisect_left` 函数**：
   - **参数**：
     - `a`：已排序的列表。
     - `x`：要查找的目标值。
     - `lo`：搜索范围的起始索引，默认为 0。
     - `hi`：搜索范围的结束索引，默认为 `len(a)`。
     - `key`：可选的键函数，用于自定义排序顺序。
   - **逻辑**：
     - 检查 `lo` 是否非负。
     - 如果 `hi` 为 `None`，则设置 `hi` 为 `len(a)`。
     - 使用二分查找算法找到目标值 `x` 应该插入的位置。
     - 如果 `key` 为 `None`，直接比较 `a[mid]` 和 `x`。
     - 如果 `key` 不为 `None`，比较 `key(a[mid])` 和 `x`。
2. **定义 `binary_search` 函数**：
   - 使用 `bisect_left` 找到目标值在已排序列表中第一次出现的位置。
   - 检查目标值是否存在于列表中：
     - 如果 `index` 不等于列表的长度且 `arr[index]` 等于目标值，返回 `index`。
     - 否则，返回 -1。

自定义键函数示例

假设你有一个包含元组的列表，并且你希望根据元组的第二个元素进行二分查找：

```python
def binary_search_with_key(arr, target, key):
    index = bisect_left(arr, target, key=key)
    if index != len(arr) and key(arr[index]) == target:
        return index  # 返回目标值的索引
    else:
        return -1  # 如果未找到目标值，返回 -1

# 示例
arr = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
target = 'c'
result = binary_search_with_key(arr, target, key=lambda x: x[1])
print(f"Target {target} found at index {result}")
```

- **输入**：

```
arr = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
target = 'c'
```

- **输出**：

```
Target c found at index 2
```

总结

二分查找是一种高效的查找算法，适用于已排序的数据集。你可以使用 `bisect` 模块中的 `bisect_left` 函数来快速实现二分查找，也可以手动实现以学习算法的细节。

进一步优化

如果你的 `key` 函数比较复杂，可以考虑使用 `functools.cmp_to_key` 来定义一个比较函数。这样可以更灵活地处理复杂的比较逻辑。

使用 `functools.cmp_to_key` 的示例

```python
from bisect import bisect_left
from functools import cmp_to_key

def compare_items(x, y):
 return (x[1] > y[1]) - (x[1] < y[1])

def binary_search_with_key(arr, target, key):
 # 找到目标值应该插入的位置
 index = bisect_left(arr, target, key=cmp_to_key(key))

```

```
    # 检查是否找到了目标值
    if index < len(arr) and key(arr[index], (0, target)) == 0:
        return index  # 返回目标值的索引
    else:
        return -1  # 如果未找到目标值，返回 -1

# 示例
arr = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
target = 'c'
result = binary_search_with_key(arr, target, key=compare_items)
print(f"Target {target} found at index {result}")
```

详细解释

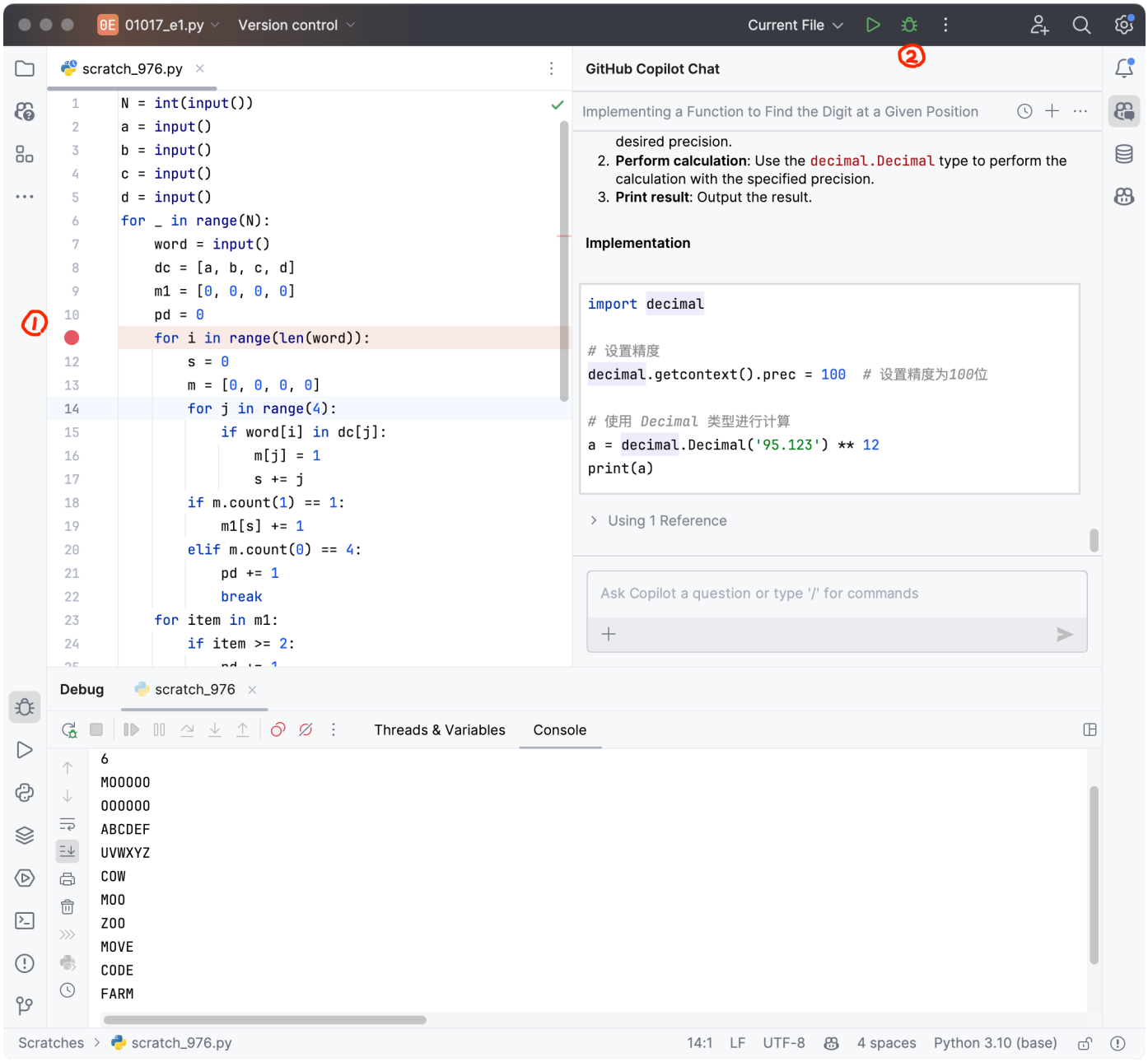
1. **`compare_items` 函数**：
   - 定义一个比较函数 `compare_items`，用于比较两个元组的第二个元素。
2. **`cmp_to_key` 函数**：
   - 将 `compare_items` 转换为 `key` 函数，传递给 `bisect_left`。
3. **`if index < len(arr) and key(arr[index], (0, target)) == 0`**：
   - 使用 `key` 函数比较 `arr[index]` 和 `(0, target)`，确保它们的第二个元素相等。

调试程序

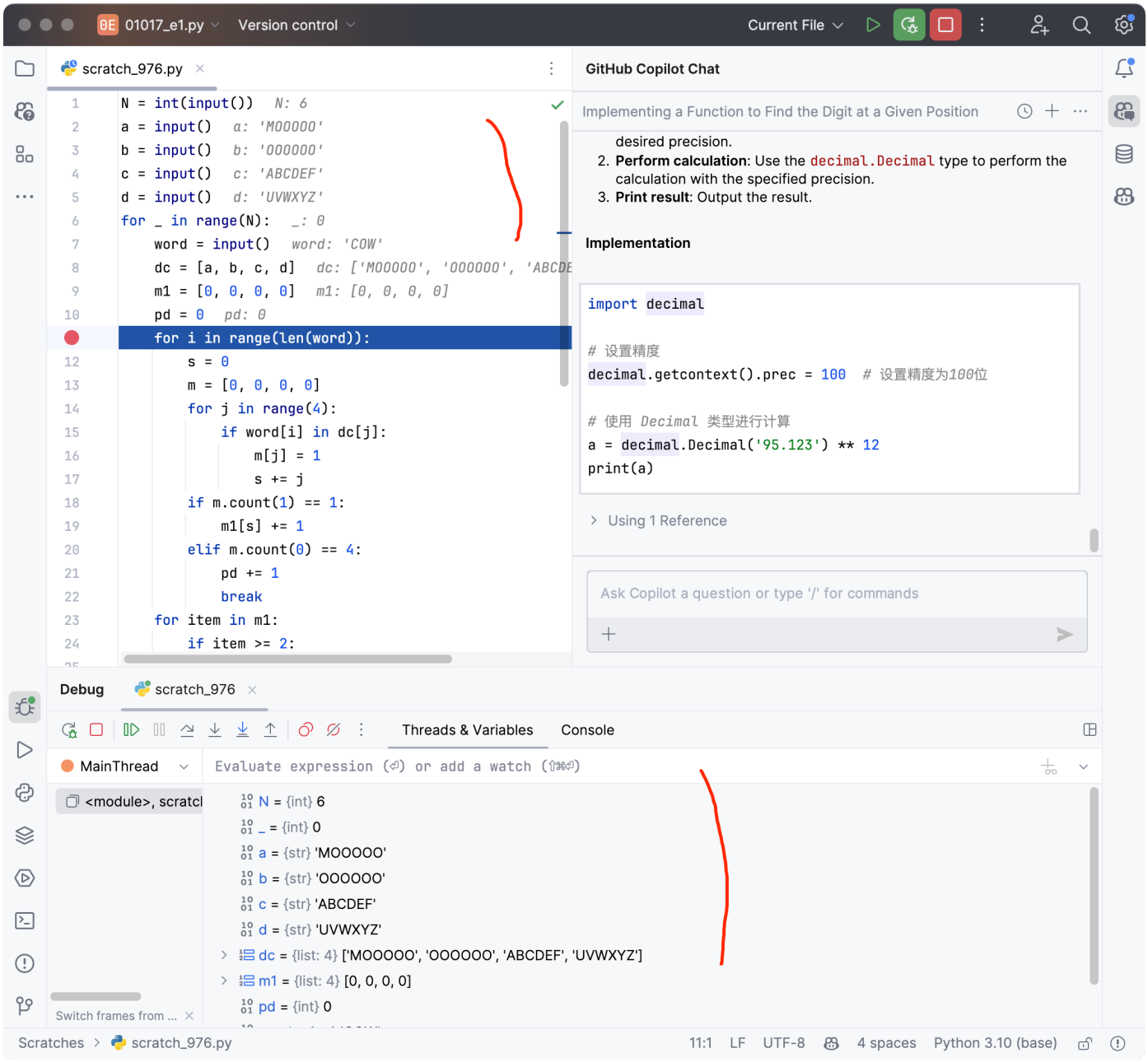
https://pythontutor.com 很好用，适合还不会用Pycharm调试工具的，当然后者也好用。另外就是print变量输出。

1 Pycharm调试

在行号处点击，设置端点；然后点击右上角绿色小虫子debug模式运行。

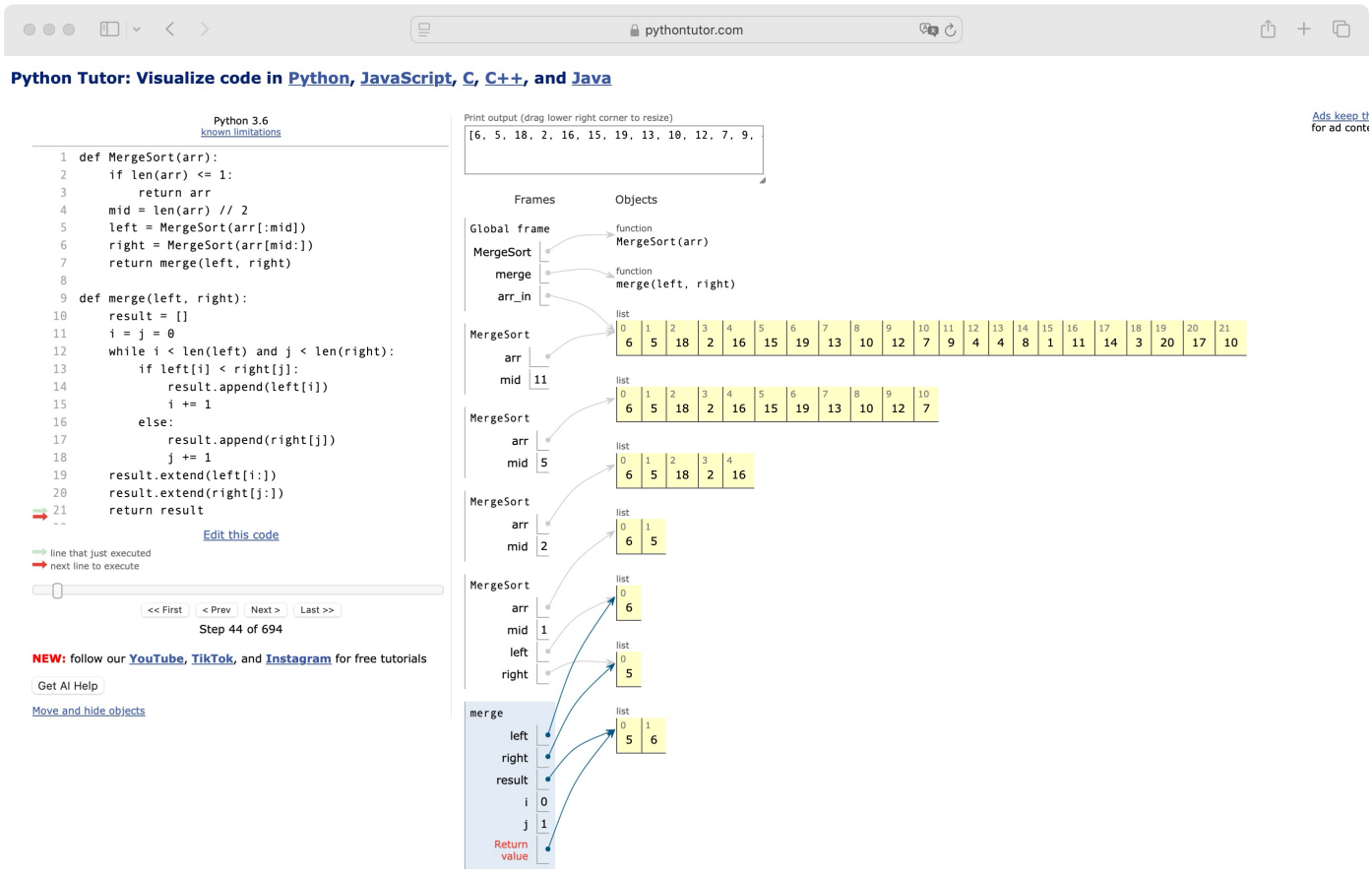


debug运行后，停在设置了短点的语句，变量中的值都显示出来。



2 pythontutor可视化运行

递归程序运行过程，不容易理解。https://pythontutor.com，完美展示 归并排序 的递归过程。



3 testing_code.py

https://github.com/GMyhf/2024fall-cs101/blob/main/code/testing_code.py

```python
# ZHANG Yuxuan
import subprocess
import difflib
import os
import sys

def test_code(script_path, infile, outfile):
    command = ["python", script_path]  # 使用Python解释器运行脚本
    with open(infile, 'r') as fin, open(outfile, 'r') as fout:
        expected_output = fout.read().strip()
        # 启动一个新的子进程来运行指定的命令
        process = subprocess.Popen(command, stdin=fin, stdout=subprocess.PIPE)
        actual_output, _ = process.communicate()
        if actual_output.decode().strip() == expected_output:
            return True
```

```
        else:
            print(f"Output differs for {infile}:")
            diff = difflib.unified_diff(
                expected_output.splitlines(),
                actual_output.decode().splitlines(),
                fromfile='Expected', tofile='Actual', lineterm=''
            )
            print('\n'.join(diff))
            return False


if __name__ == "__main__":
    # 检查命令行参数的数量
    if len(sys.argv) != 2:
        print("Usage: python testing_code.py <filename>")
        sys.exit(1)

    # 获取文件名
    script_path = sys.argv[1]

    #script_path = "class.py"  # 你的Python脚本路径
    #test_cases = ["d.in"]  # 输入文件列表
    #expected_outputs = ["d.out"]  # 预期输出文件列表
    # 获取当前目录下的所有文件
    files = os.listdir('.')

    # 筛选出 .in 和 .out 文件
    test_cases = [f for f in files if f.endswith('.in')]
    test_cases = sorted(test_cases, key=lambda x: int(x.split('.')[0]))
    #print(test_cases)
    expected_outputs = [f for f in files if f.endswith('.out')]
    expected_outputs = sorted(expected_outputs, key=lambda x: int(x.split('.')[0]))
    #print(expected_outputs)

    for infile, outfile in zip(test_cases, expected_outputs):
        if not test_code(script_path, infile, outfile):
            break

```

二、常见区间问题

一文读懂五类常见区间问题，https://zhuanlan.zhihu.com/p/446371757

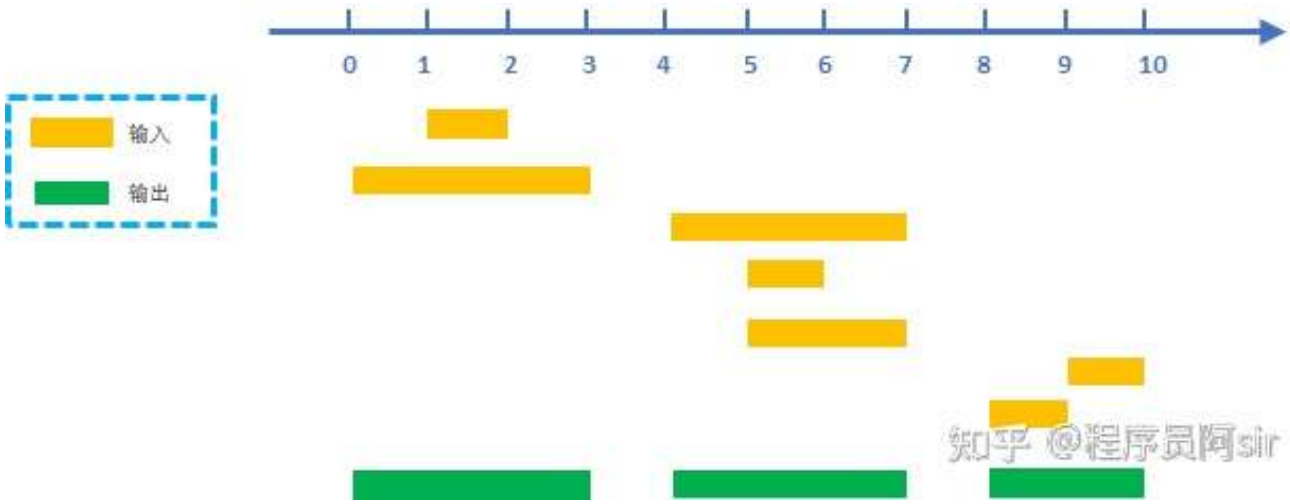
1 区间合并

1.1 题意描述

区间合并问题大概**题意**就是：

给出一堆区间，要求**合并**所有**有交集的区间**（端点处相交也算有交集）。最后问合并之后的**区间**。

如下图所示：



区间合并问题示例：合并结果包含3个区间

1.2 解题步骤

【步骤一】：按照区间**左端点**从小到大排序。

【步骤二】：维护前面区间中最右边的端点为ed。从前往后枚举每一个区间，判断是否应该将当前区间视为新区间。

假设当前遍历到的区间为第i个区间 $[l_i, r_i]$，有以下两种情况：

- $l_i \leq ed$：说明当前区间与前面区间**有交集**。因此**不需要**增加区间个数，但需要设置 $ed = max(ed, r_i)$。
- $l_i > ed$: 说明当前区间与前面**没有交集**。因此**需要**增加区间个数，并设置 $ed = max(ed, r_i)$。

LeetCode 56.合并区间（实例）

https://leetcode.cn/problems/merge-intervals/

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]` 。请你合并所有重叠的区间，并返回 *一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间* 。

示例 1：

```
输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
输出：[[1,6],[8,10],[15,18]]
解释：区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].
```

示例 2：

```
输入：intervals = [[1,4],[4,5]]
输出：[[1,5]]
解释：区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

提示：

- `1 <= intervals.length <= 10^4`
- `intervals[i].length == 2`
- `0 <= starti <= endi <= 10^4`

```python
from typing import List
import sys

class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        # 对区间进行排序
        intervals.sort()

        res = []
        st, ed = -sys.maxsize, -sys.maxsize

        for v in intervals:
            if ed == -sys.maxsize:
                st, ed = v[0], v[1]
            elif v[0] <= ed:
                ed = max(v[1], ed)
            elif v[0] > ed:
                res.append([st, ed])
                st, ed = v[0], v[1]

        if ed != -sys.maxsize:
            res.append([st, ed])

        return res
```

这里有几个需要注意的地方：

- `-sys.maxsize` 用于表示最小整数值。
- 类型注解（如 `List[List[int]]`）是可选的，但有助于提高代码的可读性和类型检查工具的效率。

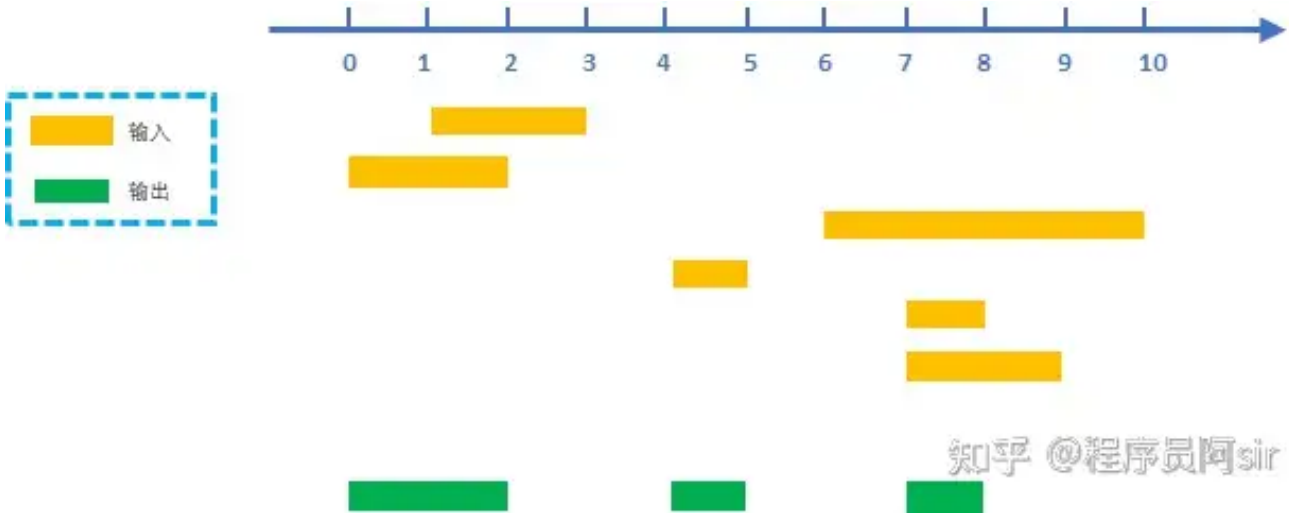
2 选择不相交区间

2.1 题意描述

选择不相交区间问题大概题意就是：

给出一堆区间，要求选择**尽量多**的区间，使得这些区间**互不相交**，求可选取的区间的**最大数量**。这里端点相同也算有重复。

如下图所示：



选择不相交区间问题示例：结果包含3个区间

2.2 解题步骤

【步骤一】：按照区间**右端点**从小到大排序。

【步骤二】：从前往后依次枚举每个区间。

假设当前遍历到的区间为第i个区间 $[l_i, r_i]$，有以下两种情况：

- $l_i \leq ed$：说明当前区间与前面区间有交集。因此直接跳过。
- $l_i > ed$: 说明当前区间与前面没有交集。因此选中当前区间，并设置 $ed = r_i$。

LeetCode 435.无重叠区间（实例）

https://leetcode.cn/problems/non-overlapping-intervals/

给定一个区间的集合 `intervals` ，其中 `intervals[i] = [starti, endi]` 。返回 *需要移除区间的最小数量，使剩余区间互不重叠* 。

注意 只在一点上接触的区间是 **不重叠的**。例如 `[1, 2]` 和 `[2, 3]` 是不重叠的。

示例 1:

```
输入: intervals = [[1,2],[2,3],[3,4],[1,3]]
输出: 1
解释: 移除 [1,3] 后，剩下的区间没有重叠。
```

示例 2:

```
输入: intervals = [ [1,2], [1,2], [1,2] ]
输出: 2
解释: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。
```

示例 3:

```
输入: intervals = [ [1,2], [2,3] ]
输出: 0
解释: 你不需要移除任何区间，因为它们已经是无重叠的了。
```

提示:

- `1 <= intervals.length <= 10^5`
- `intervals[i].length == 2`
- `-5 * 10^4 <= starti < endi <= 5 * 10^4`

```python
from typing import List
import sys

class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        # 按照右端点从小到大排序
        intervals.sort(key=lambda x: x[1])

        res = 0
        ed = -sys.maxsize

        for v in intervals:
            if ed <= v[0]:
                res += 1
                ed = v[1]

        return len(intervals) - res
```

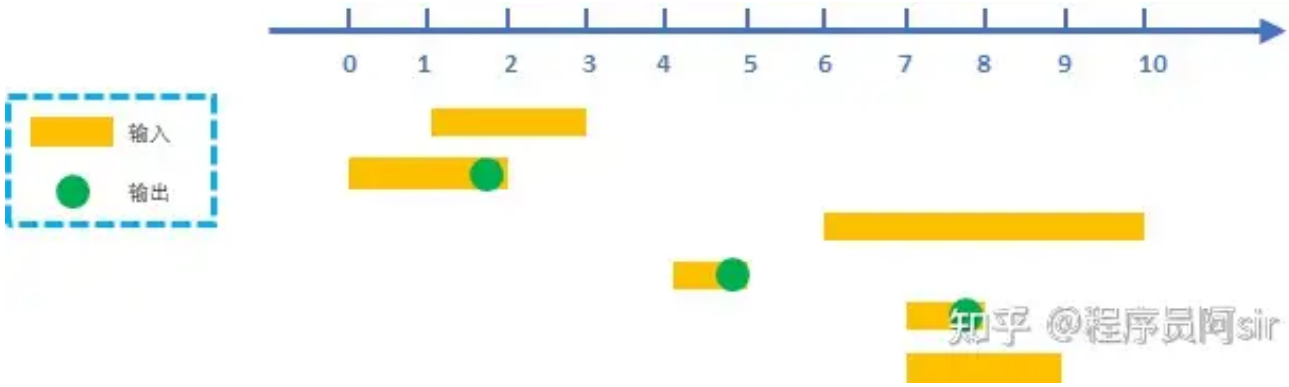
3 区间选点问题

3.1 题意描述

区间选点问题大概题意就是：

给出一堆区间，取**尽量少**的点，使得每个区间内**至少有一个点**（不同区间内含的点可以是同一个，位于区间端点上的点也算作区间内）。

如下图所示：



区间选点问题示例，最终至少选择3个点

这个题可以转化为上一题的**求最大不相交区间**的数量。

对于这些**最大的不相交区间**，肯定是每个区间都需要选出一个点。而其他的区间都是和这些选出的区间有重复的，我们只需要把点的位置选在**重合**的部分即可。

也可以换一种思路：

我们将区间按照**右端点**从小到大排序，这时我们应该尽量选择**当前区间最右边的点**。

因为最右边的点可能和下面的其他区间重复，所以至少不比选择区间靠前位置的点差。

所以，最后的解法与选择不相交区间问题解法完全一样。

3.2 解题步骤

【步骤一】：按照区间右端点从小到大排序。

【步骤二】：从前往后依次枚举每个区间。

假设当前遍历到的区间为第i个区间 $[l_i, r_i]$，有以下两种情况：

- $l_i \leq ed$：说明当前区间与前面区间有交集，前面已经选点了。因此直接跳过。
- $l_i > ed$: 说明当前区间与前面没有交集。因此选中当前区间，并设置 $ed = r_i$。

LeetCode 452. 用最少量的箭引爆气球（实例）

https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/

有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 `points` ，其中`points[i] = [xstart, xend]` 表示水平直径在 `xstart` 和 `xend`之间的气球。你不知道气球的确切 y 坐标。

一支弓箭可以沿着 x 轴从不同点 **完全垂直** 地射出。在坐标 `x` 处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 `xstart ≤ x ≤ xend`，则该气球会被 **引爆** 。可以射出的弓箭的数量 **没有限制** 。 弓箭一旦被射出之后，可以无限地前进。

给你一个数组 `points` ， *返回引爆所有气球所必须射出的 **最小** 弓箭数* 。

示例 1：

```
输入：points = [[10,16],[2,8],[1,6],[7,12]]
输出：2
解释：气球可以用2支箭来爆破：
-在x = 6处射出箭，击破气球[2,8]和[1,6]。
-在x = 11处发射箭，击破气球[10,16]和[7,12]。
```

示例 2：

```
输入：points = [[1,2],[3,4],[5,6],[7,8]]
输出：4
解释：每个气球需要射出一支箭，总共需要4支箭。
```

示例 3：

```
输入：points = [[1,2],[2,3],[3,4],[4,5]]
输出：2
解释：气球可以用2支箭来爆破：
- 在x = 2处发射箭，击破气球[1,2]和[2,3]。
- 在x = 4处射出箭，击破气球[3,4]和[4,5]。
```

提示:

- `1 <= points.length <= 105`
- `points[i].length == 2`
- `-231 <= xstart < xend <= 231 - 1`

```python
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
```

```
        # 按照右端点从小到大排序
        points.sort(key=lambda x: x[1])

        res = 0
        ed = -sys.maxsize

        for v in points:
            if ed < v[0]:
                res += 1
                ed = v[1]

        return res
```

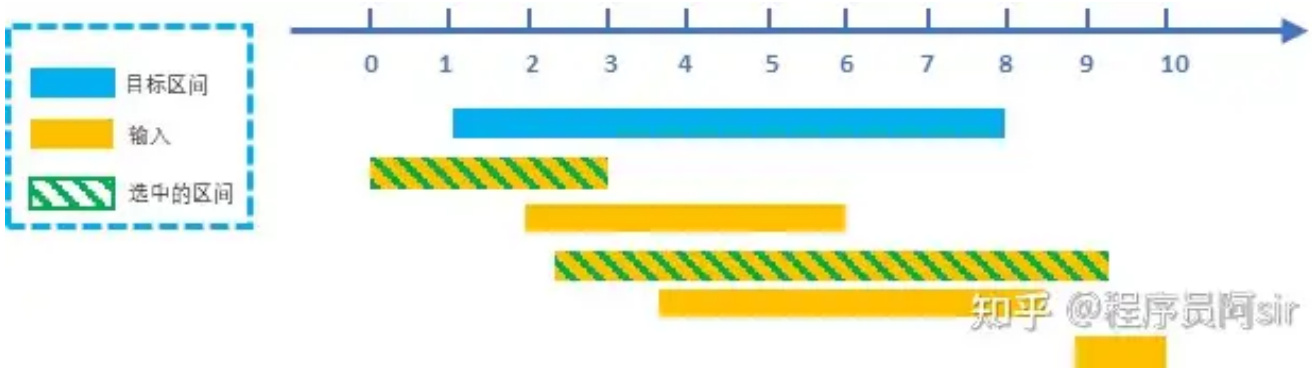
4 区间覆盖问题

4.1 题意描述

区间覆盖问题大概题意就是：

给出一堆区间和一个目标区间，问最少选择多少区间可以**覆盖**掉题中给出的这段目标区间。

如下图所示：



区间覆盖问题示例，最终至少选择2个区间才能覆盖目标区间

4.2. 解题步骤

【步骤一】：按照区间左端点从小到大排序。

步骤二】：**从前往后**依次枚举每个区间，在所有能覆盖当前目标区间起始位置start的区间之中，选择**右端点**最大的区间。

假设右端点最大的区间是第i个区间，右端点为 r_i。

最后将目标区间的start更新成r_i

LeetCode 1024. 视频拼接（实例）

https://leetcode.cn/problems/video-stitching/

你将会获得一系列视频片段，这些片段来自于一项持续时长为 `time` 秒的体育赛事。这些片段可能有所重叠，也可能长度不一。

使用数组 `clips` 描述所有的视频片段，其中 `clips[i] = [starti, endi]` 表示：某个视频片段开始于 `starti` 并于 `endi` 结束。

甚至可以对这些片段自由地再剪辑：

- 例如，片段 `[0, 7]` 可以剪切成 `[0, 1] + [1, 3] + [3, 7]` 三部分。

我们需要将这些片段进行再剪辑，并将剪辑后的内容拼接成覆盖整个运动过程的片段（`[0, time]`）。返回所需片段的最小数目，如果无法完成该任务，则返回 `-1` 。

示例 1：

```
输入：clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], time = 10
输出：3
解释：
选中 [0,2], [8,10], [1,9] 这三个片段。
然后，按下面的方案重制比赛片段：
将 [1,9] 再剪辑为 [1,2] + [2,8] + [8,9] 。
现在手上的片段为 [0,2] + [2,8] + [8,10]，而这些覆盖了整场比赛 [0, 10]。
```

示例 2：

```
输入：clips = [[0,1],[1,2]], time = 5
输出：-1
解释：
无法只用 [0,1] 和 [1,2] 覆盖 [0,5] 的整个过程。
```

示例 3：

```
输入：clips = [[0,1],[6,8],[0,2],[5,6],[0,4],[0,3],[6,7],[1,3],[4,7],[1,4],[2,5],[2,6],
[3,4],[4,5],[5,7],[6,9]], time = 9
输出：3
解释：
选取片段 [0,4], [4,7] 和 [6,9] 。
```

提示：

- `1 <= clips.length <= 100`
- `0 <= starti <= endi <= 100`
- `1 <= time <= 100`

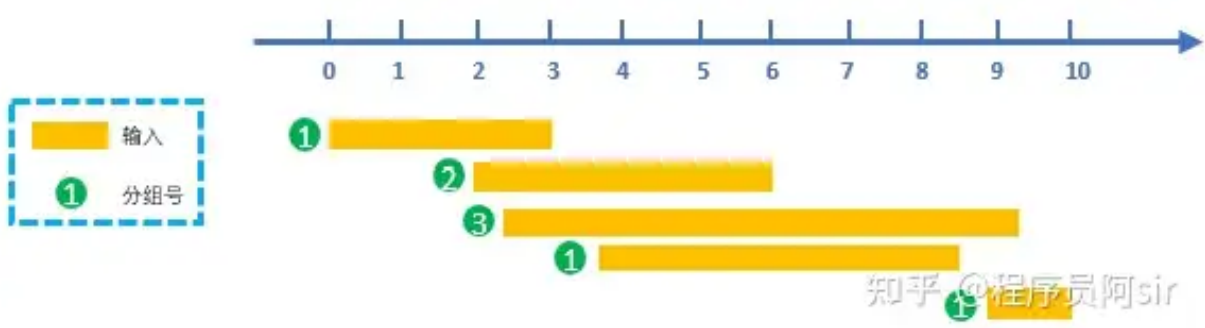
```python
from typing import List

class Solution:
    def videoStitching(self, clips: List[List[int]], time: int) -> int:
        # 对 clips 按起点升序排序
        clips.sort()

        st, ed = 0, time
        res = 0

        i = 0
        while i < len(clips) and st < ed:
            maxR = 0
            # 找到所有起点小于等于 st 的片段，并记录这些片段的最大终点 maxR
            while i < len(clips) and clips[i][0] <= st:
                maxR = max(maxR, clips[i][1])
                i += 1

            if maxR <= st:
                # 无法继续覆盖
                return -1

            # 更新 st 为 maxR，并增加结果计数
            st = maxR
            res += 1

            if maxR >= ed:
                # 已经覆盖到终点
                return res

        # 如果没有成功覆盖到终点
        return -1
```

5 区间分组问题

5.1 题意描述

区间分组问题大概题意就是：给出一堆区间，问最少可以将这些区间分成多少组使得每个组内的区间互不相交。

如下图所示：



区间分组问题示例，最少分成3个组

5.2. 解题步骤

【步骤一】：按照区间左端点从小到大排序。

【步骤二】：从**前往后**依次枚举每个区间，判断当前区间能否被放到某个现有组里面。

（即判断是否存在某个组的右端点在当前区间之中。如果可以，则不能放到这一组）

假设现在已经分了 m 组了，第 k 组最右边的一个点是 r_k，当前区间的范围是 $[L_i, R_i]$ 。则：

如果$L_i \leq r_k$ 则表示第 i 个区间无法放到第 k 组里面。反之，如果 $L_i > r_k$， 则表示可以放到第 k 组。

- 如果所有 m 个组里面没有组可以接收当前区间，则当前区间新开一个组，并把自己放进去。
- 如果存在可以接收当前区间的组 k，则将当前区间放进去，并更新当前组的 $r_k = R_i$。

注意：

为了能快速的找到能够接收当前区间的组，我们可以使用**优先队列 （小顶堆）**。

优先队列里面记录每个组的右端点值，每次可以在 O(1) 的时间拿到右端点中的的最小值。

NC147 主持人调度（实例）

https://www.nowcoder.com/questionTerminal/4edf6e6d01554870a12f218c94e8a299

有 n 个活动即将举办，每个活动都有开始时间与活动的结束时间，第 i 个活动的开始时间是 starti ,第 i 个活动的结束时间是 endi ,举办某个活动就需要为该活动准备一个活动主持人。

一位活动主持人在同一时间只能参与一个活动。并且活动主持人需要全程参与活动，换句话说，一个主持人参与了第 i 个活动，那么该主持人在 (starti,endi) 这个时间段不能参与其他任何活动。求为了成功举办这 n 个活动，最少需要多少名主持人。

数据范围: $1 \leq n \leq 10^5$， $-2^{32} \leq start_i \leq end_i \leq 2^{31}$

复杂度要求：时间复杂度 O(nlogn) ，空间复杂度 O(n)

示例1

输入

```
2,[[1,2],[2,3]]
```

输出

```
1
```

说明

```
只需要一个主持人就能成功举办这两个活动
```

示例2

输入

```
2,[[1,3],[2,4]]
```

输出

```
2
```

说明

```
需要两个主持人才能成功举办这两个活动
```

备注:

```
1≤n≤10^5
starti,endi在int范围内
```

```python
from typing import List
import heapq

class Solution:
    def minmumNumberOfHost(self, n: int, startEnd: List[List[int]]) -> int:
        # 按左端点从小到大排序
        startEnd.sort(key=lambda x: x[0])

        # 创建小顶堆
        q = []

        for i in range(n):
            if not q or q[0] > startEnd[i][0]: 有重叠，新建一个主持人
                heapq.heappush(q, startEnd[i][1])
            else:
                heapq.heappop(q)
                heapq.heappush(q, startEnd[i][1])  更新组内最大结束时间（就是堆顶那个数）

        return len(q)
```

链接：https://www.nowcoder.com/questionTerminal/4edf6e6d01554870a12f218c94e8a299
来源：牛客网

解法2:

将活动开始时间写入一个列表starts，进行排序。

将活动结束时间写入一个列表ends，进行排序。

每次活动开始时，需要增加一个主持人上场，每次活动结束时候可以释放一个主持人。

所以按照时间先后顺序对starts进行遍历，每次有活动开始count++，每次有活动结束count--

在count最大的时候，即是需要主持人最多的时候

```
class Solution:
    def minmumNumberOfHost(self , n , startEnd ):
        starts=[]
        ends=[]
        for start,end in startEnd:
            starts.append(start);
            ends.append(end);

        starts.sort();
        ends.sort()

        i,j,count,res=0,0,0,0
        for time in starts:
            while(i<n and starts[i]<=time):
                i+=1
                count+=1
            while(j<n and ends[j]<=time):
                j+=1
                count-=1
            if res<count:
                res=count
        return res
```

三、Python十大排序算法源码

> Logs:
>
> 2024/10/22 取自, https://github.com/GMyhf/2024spring-cs201/blob/main/code/ten_sort_algorithms.md

1 前言

经常用到各种排序算法，但是网上的Python排序源码质量参差不齐。因此结合网上的资料和个人理解，整理了一份可直接使用的排序算法Python源码。

包括：冒泡排序（Bubble Sort），插入排序（Insertion Sort），选择排序（Selection Sort），希尔排序（Shell Sort），归并排序（Merge Sort），快速排序（Quick Sort），堆排序（Heap Sort），计数排序（Counting Sort），桶排序（Bucket Sort），基数排序（Radix Sort）

2 排序算法的选取规则

选择合适的排序算法取决于多种因素，包括数据的规模、特性、性能要求、稳定性要求、内存限制等。

数据规模

小规模，通常指数据量在几千到几万个元素。冒泡排序、插入排序、选择排序。
中规模数据，通常指数据量在几万到几百万个元素。希尔排序、快速排序、归并排序。
大规模数据，通常指数据量在几百万到几亿甚至更多个元素。归并排序、快速排序、堆排序、外部排序、分布式排序。

数据特性

几乎有序：插入排序。

数据范围小：计数排序。

数据分布均匀：桶排序。

固定长度的整数或字符串：基数排序。

性能要求

高时间效率：归并排序、快速排序、堆排序。

低空间复杂度：选择排序、堆排序。

稳定性要求

需要稳定排序：归并排序、计数排序、基数排序、桶排序、插入排序、冒泡排序。

内存限制

内存有限：选择排序、堆排序。

Comparison sorts

在排序算法中，稳定性是指相等元素的相对顺序是否在排序后保持不变。换句话说，如果排序算法在排序过程中保持了相等元素的相对顺序，则称该算法是稳定的，否则是不稳定的。

对于判断一个排序算法是否稳定，一种常见的方法是观察交换操作。挨着交换（相邻元素交换）是稳定的，而隔着交换（跳跃式交换）可能会导致不稳定性。

Below is a table of comparison sorts. A comparison sort cannot perform better than O(n log n) on average.

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	$nlog^2n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.
Heapsort	$nlogn$	$nlogn$	$nlogn$	1	No	Selection	
Merge sort	$nlogn$	$nlogn$	$nlogn$	*n*	Yes	Merging	Highly parallelizable (up to *O*(log *n*) using the Three Hungarian's Algorithm)
Timsort	*n*	$nlogn$	$nlogn$	*n*	Yes	Insertion & Merging	Makes *n-1* comparisons when the data is already sorted or reverse sorted.
Quicksort	$nlogn$	$nlogn$	n^2	$logn$	No	Partitioning	Quicksort is usually done in-place with *O*(log *n*) stack space.
Shellsort	$nlogn$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Insertion sort	*n*	n^2	n^2	1	Yes	Insertion	*O*(n + d), in the worst case over sequences that have *d* inversions.
Bubble sort	*n*	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.

Highly tuned implementations use more sophisticated variants, such as Timsort (merge sort, insertion sort, and additional logic), used in Android, Java, and Python, and introsort (quicksort and heapsort), used (in variant forms) in some C++ sort implementations and in .NET.

计数排序，时间复杂度：$O(n+k)$，其中 k 是数据范围。空间复杂度：$O(k)$。稳定。适用于数据范围较小且数据分布均匀的情况。

桶排序，时间复杂度：平均情况$O(n+k)$，最坏情况$O(n^2)$。空间复杂度：$O(n+k)$。稳定。适用于数据分布均匀且已知数据范围的情况。

基数排序，时间复杂度：$O(nk)$，其中 k 是数字的位数。空间复杂度：$O(n+k)$。稳定。适用于数据范围较大但位数较少的情况，例如固定长度的整数或字符串。

3 十大排序算法的Python源码

3.1 冒泡排序$(Bubble\ Sort)$

方法： 通过重复地遍历要排序的列表，比较相邻的元素并根据需要交换它们的位置来实现排序。（比较次数多，交换次数多）
主要思想： 前后两两比较，大小顺序错误就交换位置

代码思路：

1. 比较相邻元素，如果前者大于后者，就交换位置。
2. 从队首到队尾，每一对相邻元素都重复上述步骤，最后一个元素为最大元素。
3. 针对前n-1个元素重复。

```python
def BubbleSort(arr):
    for i in range(len(arr) - 1):
        for j in range(len(arr) - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]
    print(arr_in)
    arr_out = BubbleSort(arr_in)
    print(arr_out)
```

时间复杂度：平均和最坏情况$O(n^2)$，最好情况$O(n)$
空间复杂度：$O(1)$
稳定排序。适用于小规模数据或几乎有序的数据。

改进后的冒泡排序是对原始冒泡排序的一种优化。原始冒泡排序的基本思想是依次比较相邻的两个元素，如果它们的顺序错误就交换它们，直到没有需要交换的元素为止。这样的算法效率较低，因为即使序列已经有序，它仍然需要进行多轮的比较和交换。

改进后的冒泡排序通过增加一个标志位来优化。在每一轮比较中，如果没有发生任何交换，说明序列已经有序，不需要再进行后续的比较，因此可以提前结束排序过程。

改进后的冒泡排序实现如下所示：

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # 标记是否发生了交换
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # 交换元素
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # 如果没有发生交换，说明数组已经排序完成
        if not swapped:
            break
    return arr
```

在这个改进后的冒泡排序算法中，如果在一轮比较中没有发生任何交换，就将标志位 `swapped` 设置为 `False`，并提前跳出循环，从而减少了不必要的比较次数，提高了效率。

3.2 选择排序$(Selection\ Sort)$

方法：在无序区找到最小的元素放到有序区的队尾（比较次数多，交换次数少）
主要思想：水果摊挑苹果，先选出最大的，再选出次大的，直到最后。
选择是对冒泡的优化，比较一轮只交换一次数据。

代码思路：

1. 找到无序待排序列中最小的元素，和第一个元素交换位置。
2. 剩下的待排无序序列（2-n）选出最小的元素，和第二个元素交换位置。
3. 直到最后选择完成。

```python
def SelectSort(arr):
    for i in range(len(arr)):
        minIndex = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[minIndex]:
                minIndex = j
        arr[i], arr[minIndex] = arr[minIndex], arr[i]
    return arr

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]
    print(arr_in)
    arr_out = SelectSort(arr_in)
    print(arr_out)
```

时间复杂度：$O(n^2)$
空间复杂度：$O(1)$
非稳定排序

3.3 插入排序$(Insertion\ Sort)$

方法：把无序区的第一个元素插入到有序区的合适位置（比较次数少，交换次数多）
主要思想：扑克牌打牌时的插入思想，逐个插入到前面的有序数中。

代码思路：

1. 选择待排无序序列的第一个元素作为有序数列的第一个元素。
2. 把第2个元素到最后一个元素看做无序待排序列。
3. 依次从待排无序序列取出每一个元素，与有序序列的每个元素比较（从右向左扫描），符合条件交换元素位置。

```python
def InsertSort(arr):
    for i in range(1, len(arr)):
        for j in range(i, 0, -1):
            if arr[j] < arr[j - 1]:
                arr[j], arr[j - 1] = arr[j - 1], arr[j]
    return arr

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]
    print(arr_in)
    arr_out = InsertSort(arr_in)
    print(arr_out)
```

时间复杂度：$O(n^2)$
空间复杂度：$O(1)$
稳定排序

上面代码并没有在找到正确位置后立即停止循环，而是一直循环直到内部的 for 循环完成。

改进后的插入排序应该在找到正确位置后立即停止循环。要实现这一点，可以在内部的 for 循环中添加一个判断条件来判断是否需要继续交换。如果当前元素已经大于（或等于）前一个元素，就可以停止内部的循环了。

下面是一个改进的插入排序版本：

```python
def InsertSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]
    print(arr_in)
    arr_out = InsertSort(arr_in)
    print(arr_out)
```

这个版本的插入排序算法在找到正确位置后会立即停止内部的循环，从而提高了效率。

3.4 希尔排序$(Shell\ Sort)$

希尔排序是插入排序的一种更高效的改进版本，其核心思想是将待排序数组分割成若干个子序列，然后对各个子序列进行插入排序，最后再对整个序列进行一次插入排序。希尔排序的关键在于选择合适的间隔序列，以保证最终的排序效率。

代码思路：

1. 选择一个增量序列t1，t2，...，tk，其中ti>tj，tk=1。
2. 按增量序列个数k，对序列进行k趟排序。
3. 每趟排序，根据对应的增量ti，将待排序序列分割成若干长度为m的子序列，分别对各子表进行直接插入排序。仅增量因子为1时，整个序列作为一个表来处理，表长度即为整个序列的长度。

```python
def ShellSort(arr):
    n = len(arr)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]
    print(arr_in)
    arr_out = ShellSort(arr_in)
    print(arr_out)
```

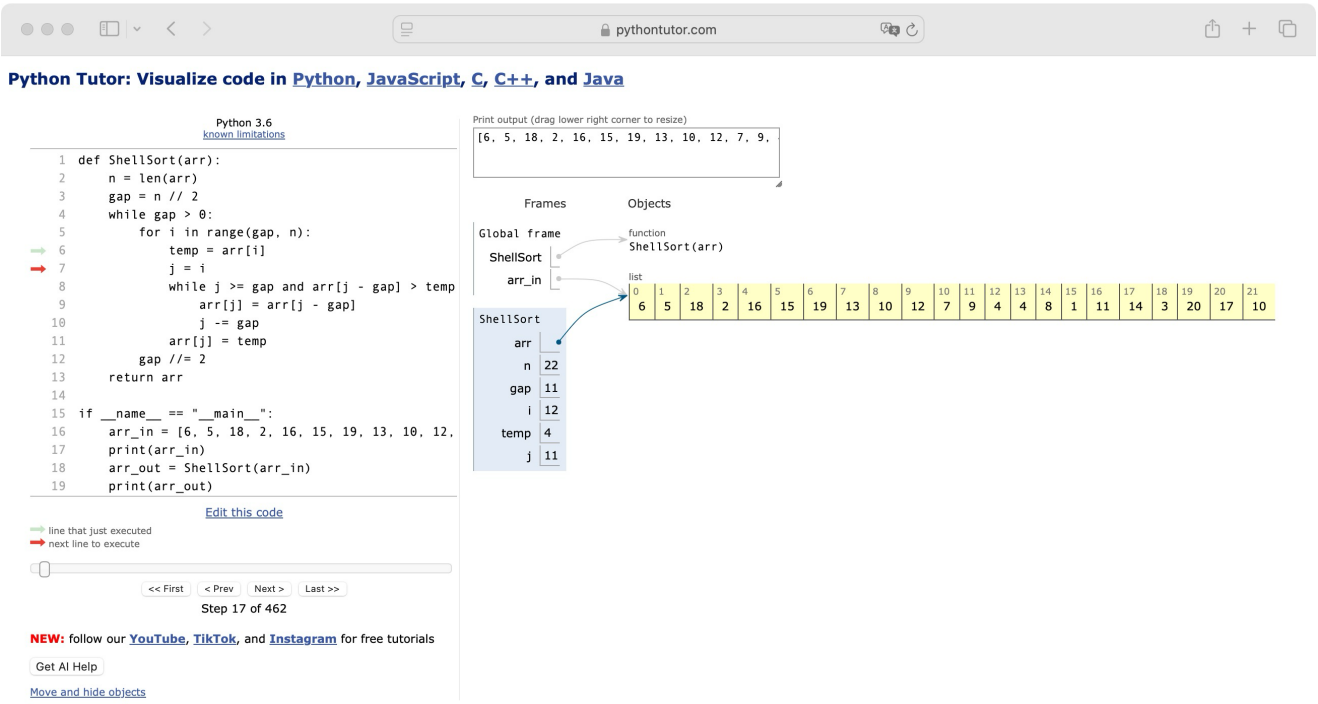
空间复杂度：$O(1)$
非稳定排序

希尔排序的时间复杂度取决于所使用的增量序列。没有一个统一的最佳增量序列，不同的增量序列会导致不同的时间复杂度。然而，通常情况下，希尔排序的时间复杂度可以描述如下：

- 最坏情况时间复杂度：$O(n^2)$，这通常发生在某些特定的增量序列上。
- 平均情况时间复杂度：根据不同的增量序列，希尔排序的平均时间复杂度可以有很大的变化，但一般认为其优于简单的插入排序，大约在$O(n^{1.25})$到$O(n^{1.6})$之间。
- 最佳情况时间复杂度：如果数组已经是有序的，或者接近有序，那么希尔排序的时间复杂度可以接近线性，即$O(n)$。

值得注意的是，对于一些特定的增量序列，希尔排序的时间复杂度可以更接近于$O(nlogn)$，但这并不普遍。因此，希尔排序对于小到中等规模的数据集是一个不错的选择，但对于非常大的数据集，可能不如快速排序或归并排序等算法高效。希尔排序的空间复杂度为$O(1)$，因为它是一种原地排序算法，不需要额外的存储空间。

https://pythontutor.com 很好用，适合还不会用Pycharm调试工具的，当然后者也好用。另外就是print变量输出。



3.5 归并排序$(Merge\ Sort)$

归并排序采用分治法，将待排序数组分成若干个子序列，分别进行排序，然后再合并已排序的子序列，直到整个序列都排好序为止。

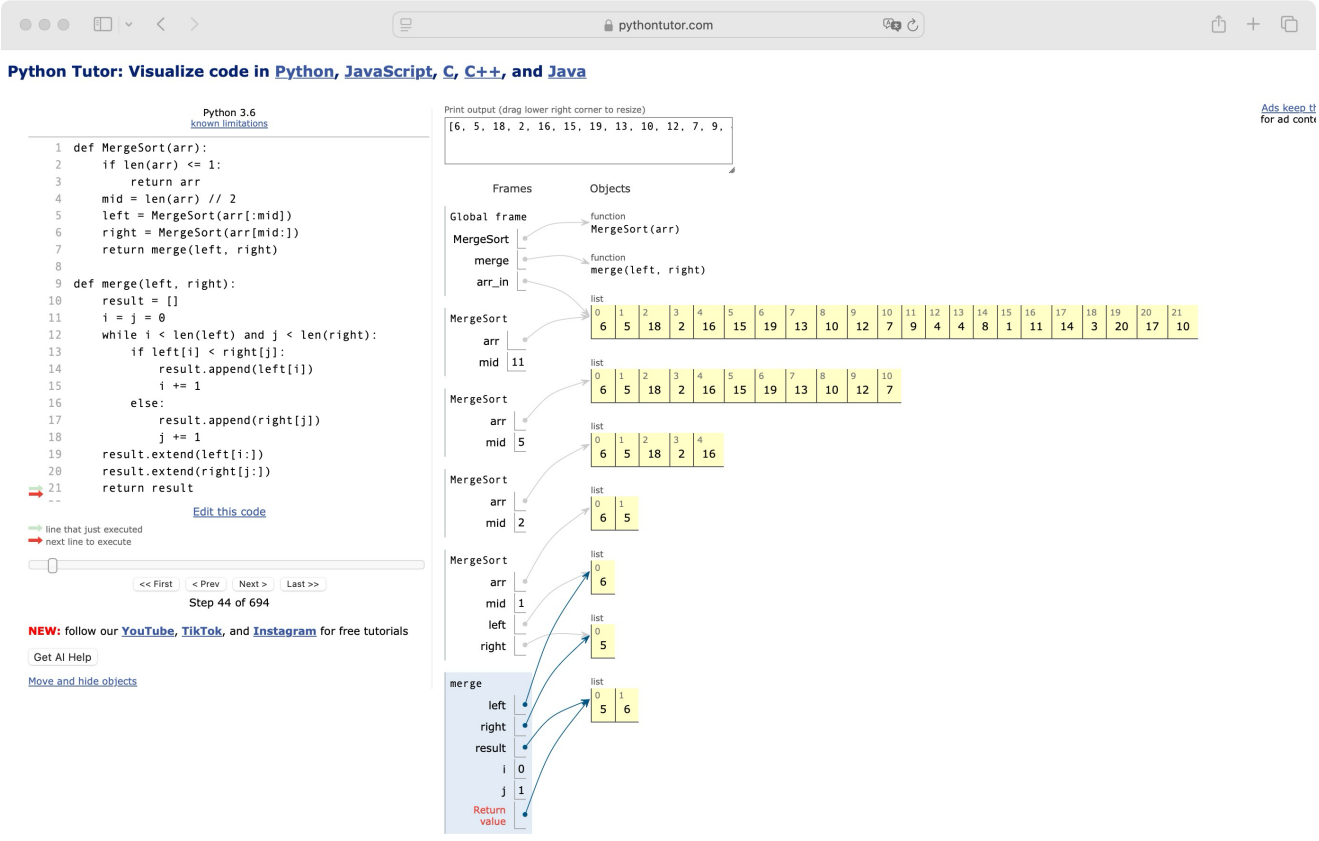
代码思路：

1. 将待排序数组分成左右两个子序列，递归地对左右子序列进行归并排序。
2. 将两个已排序的子序列合并成一个有序序列。

```
def MergeSort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = MergeSort(arr[:mid])
    right = MergeSort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
```

```
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20,
17, 10]
    print(arr_in)
    arr_out = MergeSort(arr_in)
    print(arr_out)
```

时间复杂度：O(n log n)
空间复杂度：O(n)
稳定排序

> 递归程序运行过程，不容易理解。https://pythontutor.com，完美展示 归并排序 的递归过程。
>
> 
>

3.6 快速排序$(Quick\ Sort)$

快速排序是一种高效的排序算法，采用分治法的思想，通过将数组分割成较小的子数组，然后分别对子数组进行排序，最终将数组整合成有序序列。

代码思路：

1. 选择数组中的一个元素作为基准（pivot）。
2. 将数组分割成两个子数组，使得左子数组中的所有元素都小于基准，右子数组中的所有元素都大于基准。
3. 对左右子数组递归地进行快速排序。

```python
def QuickSort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]  # Choose the first element as the pivot
        left = [x for x in arr[1:] if x < pivot]
        right = [x for x in arr[1:] if x >= pivot]
        return QuickSort(left) + [pivot] + QuickSort(right)

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]
    print(arr_in)
    arr_out = QuickSort(arr_in)
    print(arr_out)
```

时间复杂度：平均情况下为$O(nlogn)$，最坏情况下为$O(n^2)$（当数组已经有序时）
空间复杂度：平均情况下为$O(logn)$，最坏情况下为$O(n)$（递归调用栈的深度）
不稳定排序

如果用双指针实现，在partition函数中用两个指针 `i` 和 `j` 的方式实现。

```python
def quicksort(arr, left, right):
    if left < right:
        partition_pos = partition(arr, left, right)
        quicksort(arr, left, partition_pos - 1)
        quicksort(arr, partition_pos + 1, right)


def partition(arr, left, right):
    i = left
    j = right - 1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i

```

```python

arr = [22, 11, 88, 66, 55, 77, 33, 44]
quicksort(arr, 0, len(arr) - 1)
print(arr)

# [11, 22, 33, 44, 55, 66, 77, 88]
```

3.7 堆排序$(Heap\ Sort)$

堆排序利用了堆这种数据结构的特性，将待排序数组构建成一个二叉堆，然后对堆进行排序。

代码思路：

1. 构建一个最大堆（或最小堆），将待排序数组转换成堆。
2. 从堆顶开始，每次将堆顶元素与堆的最后一个元素交换，然后重新调整堆。
3. 重复上述步骤，直到整个堆排序完成。

```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def HeapSort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]
    print(arr_in)
    arr_out = HeapSort(arr_in)
    print(arr_out)
```

时间复杂度：O(n log n)
空间复杂度：O(1)
不稳定排序

3.8 计数排序$(Counting\ Sort)$

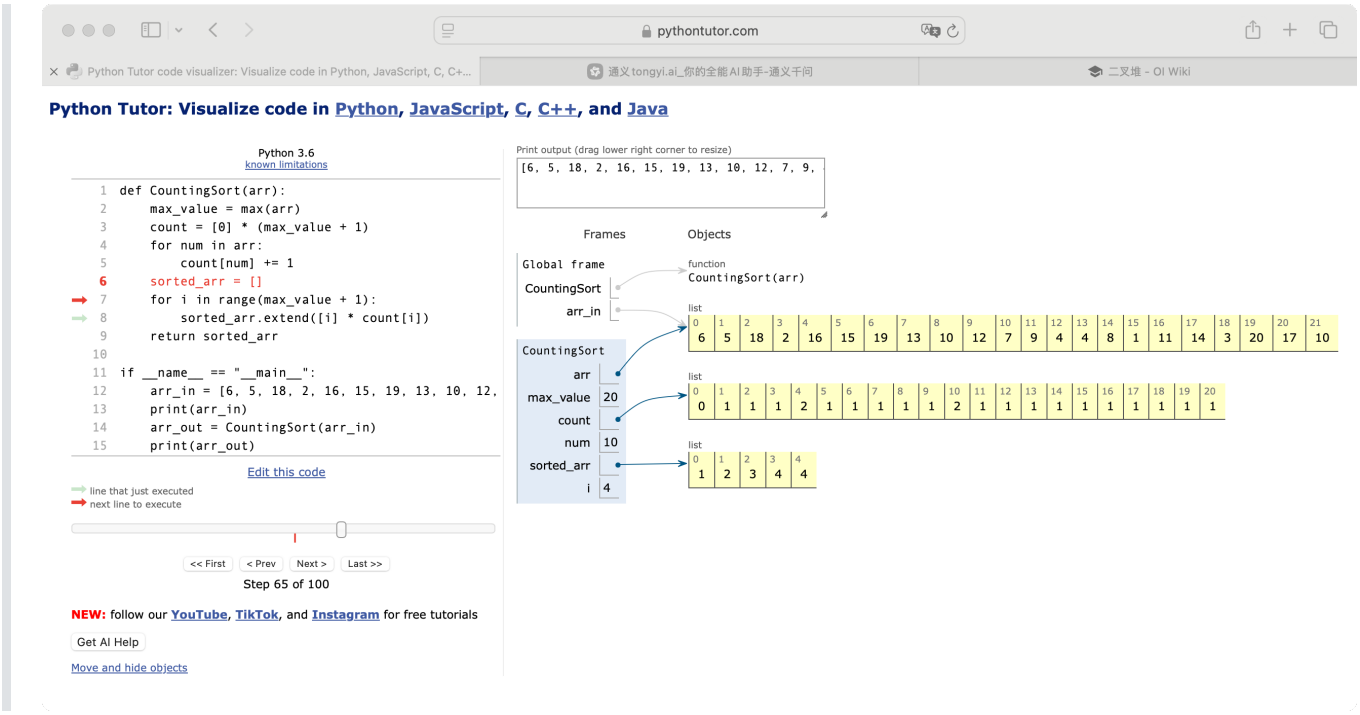
计数排序是一种非比较性的排序算法，适用于待排序数组的取值范围较小且已知的情况。该算法通过统计每个元素出现的次数，然后根据统计结果重构排序后的数组。

代码思路：

1. 统计数组中每个元素出现的次数，并存储在额外的计数数组中。
2. 根据计数数组中的统计结果，重构排序后的数组。

```python
def CountingSort(arr):
    max_value = max(arr)
    count = [0] * (max_value + 1)
    for num in arr:
        count[num] += 1
    sorted_arr = []
    for i in range(max_value + 1):
        sorted_arr.extend([i] * count[i])
    return sorted_arr

if __name__ == "__main__":
    arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]
    print(arr_in)
    arr_out = CountingSort(arr_in)
    print(arr_out)
```

时间复杂度：O(n + k)，其中n是数组的长度，k是数组中的最大值与最小值的差值
空间复杂度：O(n + k)



3.9 桶排序(*Bucket Sort*)

桶排序是一种排序算法，它假设输入是由一个随机过程产生的，该过程将元素均匀、独立地分布在[0, 1)区间上。

代码思路：

1. 创建一个定量的桶数组，并初始化每个桶为空。
2. 将每个元素放入对应的桶中。
3. 对每个非空桶进行排序。
4. 从每个桶中将排序后的元素依次取出，得到排序结果。

```python
def BucketSort(arr):
    n = len(arr)
    max_val = max(arr)
    min_val = min(arr)
    bucket_size = (max_val - min_val) / n

    buckets = [[] for _ in range(n+1)]

    for num in arr:
        index = int((num - min_val) // bucket_size)
        buckets[index].append(num)

    sorted_arr = []
    for bucket in buckets:
        sorted_arr.extend(sorted(bucket))

    return sorted_arr

if __name__ == "__main__":
```

```
    arr_in = [0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434]
    print(arr_in)
    arr_out = BucketSort(arr_in)
    print(arr_out)

#[0.1234, 0.3434, 0.565, 0.656, 0.665, 0.897]
```

时间复杂度：$O(n+k)$，其中n是数组的长度，k是桶的数量
空间复杂度：$O(n)$

3.10 基数排序$(Radix\ Sort)$

基数排序是一种非比较型整数排序算法，它通过按位处理数字来排序，通常用于处理非负整数。

基数排序是一种多关键字的排序算法，它将整数按位数切割成不同的数字，然后按每个位数分别比较。

代码思路：

1. 找出数组中最大值，并确定最大值的位数。
2. 使用计数排序或桶排序，根据当前位数进行排序。

```
def RadixSort(arr):
    max_val = max(arr)
    digit = len(str(max_val))

    for i in range(digit):
        bucket = [[] for _ in range(10)]
        for num in arr:
            bucket[num // (10 ** i) % 10].append(num)
        # for row in bucket:
        #     print(*row)
        arr = [num for sublist in bucket for num in sublist]
        # arr = []
        # for sublist in bucket:
        #     for num in sublist:
        #         arr.append(num)
        #print(arr)

    return arr

if __name__ == "__main__":
    arr_in = [170, 45, 75, 90, 802, 24, 2, 66]
    print(arr_in)
    arr_out = RadixSort(arr_in)
    print(arr_out)
```

时间复杂度：$O(nk)$，其中n是数组的长度，k是最大值的位数
空间复杂度：$O(n+k)$

四、单调栈（monotonic stack）

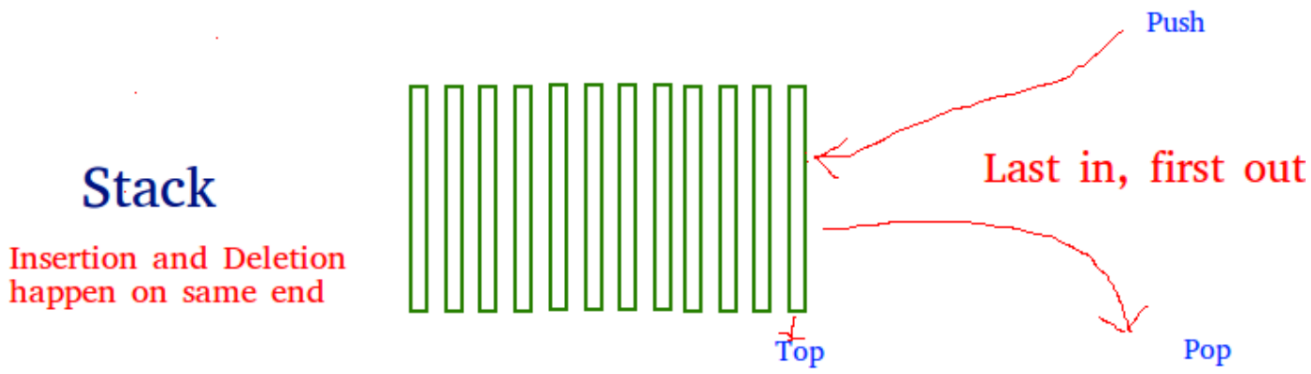
1 Stack in Python

https://www.geeksforgeeks.org/stack-in-python/

Stack in Python

Read Discuss Courses Practice

A **stack** is a linear data structure that stores items in a **Last-In/First-Out (LIFO)** or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.



The functions associated with stack are:

- **empty()** – Returns whether the stack is empty – Time Complexity: O(1)
- **size()** – Returns the size of the stack – Time Complexity: O(1)
- **top() / peek()** – Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- **push(a)** – Inserts the element 'a' at the top of the stack – Time Complexity: O(1)
- **pop()** – Deletes the topmost element of the stack – Time Complexity: O(1)

2 Monotonic Stack（单调栈）

单调栈是一种特殊的栈结构，其中的元素按照某种特定的顺序（如递增或递减）排列。在计算机科学中，单调栈常用于解决一类与数组或序列相关的优化问题，比如寻找下一个更大或更小的元素等。

单调栈的应用场景

1. **寻找下一个更大的元素**：给定一个数组，对于每个元素，找到它右边第一个比它大的元素的位置。这类问题可以使用单调递减栈来高效解决。
2. **寻找下一个更小的元素**：类似地，如果需要找到每个元素右边第一个比它小的元素的位置，则可以使用单调递增栈。
3. **直方图中的最大矩形**：这是一个经典的问题，涉及到计算直方图中最大的矩形面积，可以使用单调栈来有效求解。
4. **滑动窗口的最大值**：虽然这个问题通常使用双端队列来解决，但也可以通过单调栈的变形来处理。

单调栈的工作原理

- **入栈操作**：当一个新的元素需要加入到栈中时，根据栈的性质（递增或递减），将所有不符合条件的栈顶元素弹出，然后再将新元素压入栈中。
- **出栈操作**：通常情况下，出栈操作是自动发生的，即在执行入栈操作时，为了保持栈的单调性，会自动移除不满足条件的栈顶元素。

实现示例

这里以一个简单的例子说明如何使用单调栈来解决问题。假设我们需要找到数组 `[4, 5, 2, 25]` 中每个元素右边第一个更大的数。

```python
def next_greater_element(nums):
    stack = []
    result = [0] * len(nums)

    for i in range(len(nums)):
        # 当栈不为空且当前考察的元素大于栈顶元素时
        while stack and nums[i] > nums[stack[-1]]:
            index = stack.pop()
            result[index] = nums[i]
        # 将当前元素的索引压入栈中
        stack.append(i)

    # 对于栈中剩余的元素，它们没有更大的元素
    while stack:
        index = stack.pop()
        result[index] = -1

    return result

nums = [4, 5, 2, 25]
print(next_greater_element(nums))  # 输出: [5, 25, 25, -1]
```

在这个例子中，我们维护了一个单调递减的栈，当遇到比栈顶元素大的数时，就找到了栈顶元素的“下一个更大的数”，然后将其从栈中弹出，并记录结果。最后，对于那些在栈中没有匹配到更大数的元素，它们的结果设置为 `-1`，表示没有更大的数。

3 编程题目

04137: 最小新整数

stack, greedy, http://cs101.openjudge.cn/practice/04137/

给定一个十进制正整数n(0 < n < 1000000000)，每个数位上数字均不为0。n的位数为m。
现在从m位中删除k位(0<k < m)，求生成的新整数最小为多少?
例如: n = 9128456, k = 2, 则生成的新整数最小为12456

输入

第一行t, 表示有t组数据；
接下来t行，每一行表示一组测试数据，每组测试数据包含两个数字n, k。

输出

t行，每行一个数字，表示从n中删除k位后得到的最小整数。

样例输入

```
2
9128456 2
1444 3
```

样例输出

```
12456
1
```

```python
# 蒋子轩23工学院
def removeKDigits(num, k):
    stack = []
    for digit in num:
        while k and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)
    # 如果还未删除k位，从尾部继续删除
    while k:
        stack.pop()
        k -= 1
    return int(''.join(stack))
t = int(input())
results = []
for _ in range(t):
    n, k = input().split()
    results.append(removeKDigits(n, int(k)))
for result in results:
    print(result)
```

25702: 护林员盖房子 加强版

matrix/implementation, http://cs101.openjudge.cn/practice/21577

在一片保护林中，护林员想要盖一座房子来居住，但他不能砍伐任何树木。
现在请你帮他计算：保护林中所能用来盖房子的矩形空地的最大面积。

输入
保护林用一个二维矩阵来表示，长宽都不超过1000（即<=1000）。
第一行是两个正整数m,n，表示矩阵有m行n列。
然后是m行，每行n个整数，用1代表树木，用0表示空地。

输出
一个正整数，表示保护林中能用来盖房子的最大矩形空地面积。

26977: 接雨水

stack, dp, math, http://cs101.openjudge.cn/practice/26977/

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例：

height = [0,1,0,2,1,0,1,3,2,1,2,1]

由数组表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

输入

第一行包含一个整数n。1 <= n <= 2 * 10^4
第二行包含n个整数，相邻整数间以空格隔开。0 <= ratings[i] <= 2 * 10^5

输出

一个整数